Risk Assessment of Network Distributed Android Applications

Hossain Shahriar, Victor Clincy
Department of Computer Science
Kennesaw State University
Kennesaw, USA

Abstract

Android applications are used by millions of users to perform useful activities. Unfortunately, many legitimate and popular applications are targeted by malware authors. Malware authors repackage existing applications by injecting additional code intended to perform malicious activities. As legitimate applications are updated with new features, malware authors also keep up their effort to repackage them with malicious functionalities. Thus, it is important to validate applications for possible repackaging before their installation to safeguard end users. This paper presents the detection of repackaged Android application malware based on the use of a popular metric known as Kullback-Leibler Divergence (KLD). Our approach builds the population distribution of a legitimate and suspected repackaged malware application based on a set of Smali opcode. A high KLD value indicates that an application is dissimilar compared to an original application, hence likely a repackaged application. The approach has been validated based on real-world malware samples. The results indicate that KLD values remain high for all the malware when repackaged within a legitimate application, and hence can be used as a suitable metric for detection of new malware.

1. Introduction

Currently, Android occupies close to 80% of mobile device market share [1]. We highly depend on Android devices as well as the applications that run on the platform. In particular, many useful activities such as phone call, message sending, and game playing are performed with applications. End users rely on market place to obtain legitimate applications and install in their devices.

Unfortunately, popular Android applications are becoming the target of malware authors. In particular, there exists available open source tools that can be used to download legitimate applications, disassemble these applications, insert with additional malicious code intended to perform unauthorized activities, repackage the modified applications, and finally lure or distribute to potential victims to download the applications and install in their devices. The modified and repackaged application if installed by a victim in his/her phone, unwanted malicious activities take place without his/her knowledge.

As legitimate applications are updated with new features, malware authors also keep up their effort to repackage them with malicious functionalities. A detailed study of a large set of malware applications and characteristics revealed that most of the well-known malware samples belong to few popular legitimate applications available in the market [2]. In particular, three common types of malicious functionalities are added during repackaging (i) leveraging root level exploits to compromise Android security such as changing device password, (ii) communicating with external servers controlled by attackers (in botnet) through SMS messages sent to premium numbers, and (iii) making phone calls without user awareness. Given this, it is very important to check the possibility of a repackaged application before an application is installed. This paper proposes an offline analysis for detecting repackaged application given that we have an access to a legitimate application.

A large number of literature works have recently addressed Android malware from the perspective of classification and detection based on anomalous activities or permission sets present in suspected applications [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19]. However, very few research works have addressed the issue of detecting repackaged Android malware applications considering the availability of legitimate applications [22] [23] [24]. Existing approaches statically analyzes the assembly code. These approaches suffer from false positive warnings or generating numerous hashes based on opcode sets resulting in significant computation time. In this paper, we propose metric-based detection approach for repackaged malware. In particular, we apply a popular metric called Kullback-Leibler Divergence (KLD) [25].

A repackaged application when compared to its original version of the application is different compared to the set of available functionalities. We capture the difference by proposing a set of opcode (Smali) to compute needed population sets as part of KLD. Our proposed opcodes consider the existing knowledge from malware detection domain such as method call invocation that may send a message sneakingly as part of malware application functionality. To perform computation for missing elements of a population set, we rely on back-off smoothing algorithms.

The approach has been applied to a number of malicious and repackaged android applications and compares the legitimate version of the applications with the repackaged version. The results seem promising, KDL values between malware and known good application differs significantly.

The paper is organized as follows. Section 2 provides an overview of Android application packaging and related work. In Section 3, the proposed approach is discussed in details. Section 4 discusses the experimental evaluation. Section 5 draws the conclusions and discusses future work.

2. Overview and Related Work

2.1 Android Application Repackaging

Android application developer can package an application during release time by signing with a chosen private key that may include a developers or organization identity. The signature file stored in the package can be extracted to view the finger prints such as MD5 and SHA1 hashes using toolsets (keytool, Jarsigner). An Android application is distributed as an apk file, which can be later unzipped.

The zipped APK file includes the application code in DEX format (Dalvik Executable) which is similar to an assembly language called Smali [3]. There exists open source tools (*e.g.*, dex2jar, java decompiler, apktool) that enables anyone to disassemble an apk file and retrieve the opcode or Java source code from DEX to inspect, modify, and repackage with additional code [5] [20] [21]. Below we describe the steps to repackage an application using the apktool.

Step 1: Create a malicious application that implements an intended functionality. The attacker (malware author) then attempts to invoke the functionality either due to system level event (*e.g.*, receiving a phone call) or user level event (e.g., initiate a phone call). The malicious application is converted to an apk file format.

Step 2: Use the apktool to disassemble the malicious applications and extract the directory containing classes implementing malicious functionalities (all class files are present in the smali folder). The classes are represented as opcode format [3].

Step 3: Identify a suitable legitimate application to inject malicious activities. The legitimate application is disassembled using the same tool, and the classes that perform malicious activities are inserted in appropriate directories. If additional permission or library support is needed, the manifest file (*AndroidManifest.xml*) is modified appropriately. The modified application is assembled again with the apktool and signed with a new key.

Step 4: The repackaged application is distributed over the network and potential victims are lured to install the malware on their devices.

2.2. Related Works on Android Malware Detection

Crusselle *et al.* detect repackaged applications by computing the data dependency graph (DDG) statically for each of the methods statically [22]. The graphs are compared for similarity for a known application to identify possible deviation due to additional methods part of a repackaging malware application.

Li *et al.* propose feature hashing-based technique [23]. They first identify k-grams of various opcode sequence patterns within each basic block and consider them as features. The presence or absence of each features in dex files are encoded in a vector. All vectors obtained from files are merged to obtain the fingerprint of each application.

Zhou *et al.* computes hash values for each local unit of opcode sequence of the classes (dex files) [24]. The long opcode is handled by splitting into small units and computing hashes for each split unit. Finally, all individual hashes are combined into one hash values. This way any additional inserted code is detected not only for the overall application, but locating the files or specific instruction sets that are inserted by malware authors. The approach suffers from false positive if inserted dummy opcode does not have any negative impact.

In contrast to these efforts, our proposed technique relies on metrics and computes them at application runtime in sandbox.

We are aware of several works that classify malware applications and their detection technique. Amamra *et al.* perform a survey on malware detection approaches highlighting two broader classes of malware detection: signature and anomaly-based [7]. Porter *et al.* performed a survey on malicious characteristics for mobile device malware in 2011 [6]. Cooper *et al.* classify android malware detection techniques and comparatively identify the advantages and disadvantages including static analysis, sandboxing, and machine learning approaches [8]. Tanh *et al.* characterize malware and demonstrate what end users can do to check the presence of malware and prevent them [29].

Enck *et al.* analyzed a large set of android applications and identified dataflow, structure, and semantic patterns [9]. The dataflow patterns identify whether any sensitive data information piece should not be sent to outside (*e.g.*, IMEI, IMSI, ICC-ID). Enck *et al.* proposed a rule-based certification technique to check the presence of

undesirable properties in applications suspected as malware [10]. Batyuk *et al.* perform static analysis on binary code of android applications (after decompressing APK and decoding Java bytecode into Smali assembly language [11]. They look for the presence of APIs that may be relevant of reading sensitive information (e.g., IMEI or device identifier, IMSI or subscriber identifier, phone number, writing information to output stream). Yang *et al.* detect money stealing malware by examining the manifest file of android applications to see if billing permission is present [16]. Barrera *et al.* apply self-organizing map-based learning to cluster permission sets [12]. The study and findings cannot be suitably applied for detecting malware as both malicious and benign applications may have similar type of permissions. Similarly, Felt *et al.* compared the permission system between Google Chrome and Google Android, and performed a subjective analysis for improving permission model in general for security and user level awareness [13].

Nevertheless, detection technique of repackaged malware is still needed to identify malicious behaviors of malware, and our approach is complementary to these earlier efforts.

Several works rely on live analysis of applications running in a sandbox environment. Enck *et al.* analyze the dataflow of Android application to detect privacy leak (whether sensitive data are being transferred to third parties related to advertisement services) [14]. Blasing *et al.* also develop a sandbox to perform dynamic analysis of suspected applications in an isolated environment [15]. All these earlier efforts are complementary to our proposed approach intended to build defense in-depth against repackaged malware.

3. Proposed Detection Approach

3.1. Kullback-Leibler Divergence (KLD) Computation

The Kullback-Leibler Distance (KLD) computes the divergence or distance between two given probability distributions. Let us assume that P and Q represent two probability distributions, where $P = \{p_1, \ldots, p_n\}$ and $Q = \{q_1, \ldots, q_n\}$. Then, the KLD is defined as follows [25]:

$$KLD\ (P, Q) = \sum_{i} pi * log_2\ (p_i / q_i) \ \ldots\ (i)$$

Here, the following two constraints (Equations *(ii)* and *(iii)*) are satisfied:

$$\sum_{i} pi = 1 \ \ldots\ (ii)$$

$$\sum_{i} qi = 1 \ \ldots\ (iii)$$

We start with a hypothesis that the Kullback-Leibler Divergence (KLD) between an original and repackaged malware application should be a high number. On the other hand, the KLD among 2 legitimate applications from the same source, KLD value should be very low.

To compute the KLD between two population sets (or probability distributions) need to be defined at the beginning. We focus on a set of well-known opcode that may be common in both legitimate and repackaged malware applications. A set of opcode elements are extracted from a known legitimate application to build P set. Now, given that we have a new application (Q), we extract the similar opcode occurrence probability distribution and compute the divergence to detect possible repackaged application.

The challenge of computing KLD (P, Q) is the term $p_i * log_2\ (p_i/q_i)$. It can be rewritten as subtraction of two terms: $p_i * log_2(p_i) - p_i * log_2(q_i)$. If p_i or q_i is zero (no occurrence of a specific opcode is observed), then the term becomes infinite, which results in KLD (P, Q) to be infinite as well.

To address this issue, we propose to apply a well-known smoothing technique known as constant back-off [26]. Here, all zero probability values in both P and Q are replaced with a very negligible constant probability value and all the non-zero values are equally subtracted with the same constant value proportionally so that Equations *(ii)* and *(iii)* are still satisfied. This simple step results in two smoothed probability distributions that we denote as P' (derived from P) and Q' (derived from Q).

3.2. Elements of Population set for KLD Computation

Table 1 shows a set of Smali opcode that we propose to build population elements (f_1-f_9). We consider nine opcode: *invoke-super*, *invoke-static*, *invoke-direct*, *invoke-virtual*, *const-string*, *new-instance*, *move-result-object*, *return-object*, and *throw*. A description of the opcode is provided in Table 1.

Table 1. Description of Opcode for Population Set

Name	Smalie opcode	Description
f_1	invoke-super {parameter},methodtocall	Invokes the virtual method of the immediate parent class
f_2	invoke-static {parameters}, methodtocall	Invokes a static method with parameters.
f_3	invoke-direct {parameter},methodtocall	Invokes a method with parameters without the virtual method resolution

f_4	invoke-virtual {parameter},methodtocall	Invokes a virtual method with parameters.
f_5	const-string vx,string_id	Puts reference to a string constant identified by string_id into vx.
f_6	new-instance vx, type	Instantiates an object type and puts the reference of the newly created instance into vx
f_7	move-result-object vx	Move the result object reference of the previous method invocation into vx.
f_8	return-object vx	Return with vx object reference value.
f_9	throw vx	Throws an exception object. The reference of the exception object is in vx.

For example, *invoke-direct* opcode invokes a method specified in the second argument while supplying a set of parameter specified in the first argument. f_7 represents the *move-result-object* opcode which stores the result of an object creation to a variable. The f_9 shows an example of exception throwing opcode.

We choose these opcodes based on the literature knowledge that injected code by malware authors are mostly doing a set of operations such as sending of SMS messages to premium numbers, probing device id and sending the information over the network. All these operations require method call invocation as well as often defining constant string values (*const-string* opcode) that may store attacker supplied information such as phone number.

3.3. Example of Repackaged Malware Detection

We consider a legitimate example of Android application that is intended to display a simple message "hello world" in an Activity class as shown in Figure 1. In this example, the *onCreate()* method displays the message by accessing the *TextView* object and invoking the *setText()* method call.

```
public class HelloWorldActivity extends Activity {
   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);

      TextView text = new TextView(this);
      text.setText("Hello World, Android");
      setContentView(text);
   }
}
```

Figure 1. Example Java Code for Legitimate Application (P)

```
…
# virtual methods
.method public onCreate(Landroid/os/Bundle;)V
…
   invoke-super {p0, p1}, Landroid/app/Activity;-
>onCreate(Landroid/os/Bundle;)V
   new-instance v0, Landroid/widget/TextView;
   invoke-direct {v0, p0}, Landroid/widget/TextView;-
>(Landroid/content/Context;)V
   .local v0, text:Landroid/widget/TextView;
   const-string v1, "Hello World, Android"
   invoke-virtual {v0, v1}, Landroid/widget/TextView;-
>setText(Ljava/lang/CharSequence;)V
   invoke-virtual {p0, v0}, Lcom/test/helloworld/HelloWorldActivity;-
>setContentView(Landroid/view/View;)V
…
.end method
```

Figure 2. Example Opcode for the Legitimate Application (P)

```
public class HelloWorldActivity extends Activity {
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      TextView text = new TextView(this);
      text.setText("Hello World, Android");
      setContentView(text);
      SmsManager smsManager = SmsManager.getDefault();
      String phone = "1-900-222-3333";
      smsManager.sendTextMessage (phone, null, "sms", null, null);
   }
}
```

Figure 3. Example Java Code for a Legitimate Android Application (P)

```
# virtual methods
.method public onCreate(Landroid/os/Bundle;)V
…
   invoke-super {p0, p1}, Landroid/app/Activity;-
>onCreate(Landroid/os/Bundle;)V
   …
   new-instance v0, Landroid/widget/TextView;
   …
   invoke-direct {v0, p0}, Landroid/widget/TextView;-
>(Landroid/content/Context;)V
   .local v0, text:Landroid/widget/TextView;
   …
   const-string v1, "Hello World, Android"
   …
   invoke-virtual {v0, v1}, Landroid/widget/TextView;-
>setText(Ljava/lang/CharSequence;)V
   …
   invoke-virtual {p0, v0},
Lcom/test/helloworld/HelloWorldActivity;-
>setContentView(Landroid/view/View;)V
   …
invoke-virtual {p0}, Lcom/android/telephony/SmsManager;-
>getDefault () Landroid/content/Context;
   const-string v1, "1-900-222-3333"
   const/4 v2, 0x0
   const-string v3, "sms"
   const/4 v4, 0x0
   const/4 v5, 0x0
   invoke-static {v1, v2, v3, v4, v5},
Landroid/telephony/SmsManager;->sendTextMessage();
.end method
```

Figure 4. Example Opcode of Repackaged Android Malware (Q)

Figure 2 shows a snapshot the Smali opcode that can be obtained based on a suitable reverse engineering tool such as apktool [21]. We display and highlight the relevant opcode as part of population set (*e.g.*, invoke-super) for the *onCreate()* method call only due to space limitation.

Table 2. Occurrence of Population Element from Legitimate Application (*P*)

Opcode	Occurrence (p_i)	Smoothed (p_i')
invoke-super (f_1)	1/4	1/4 - e/5
invoke-static (f_2)	0/4	e/5
invoke-direct (f_3)	1/4	1/4 - e/5
invoke-virtual (f_4)	2/4	2/4 - 2e/5
const-string (f_5)	1/4	1/4 - e/5
new-instance (f_6)	0/4	e/5
move-result (f_7)	0/4	e/5
return-object (f_8)	0/4	e/5
throw (f_9)	0/4	e/5

Let us assume that a malware injects SMS message sending operation to a premium number right after the hello world message display operation. The added code is shown and highlighted in Figure 3. Here, the *SmsManager* object is first retrieved followed by invocation of *sendTextMessage()* method call having five arguments including a premium phone number (1-900-222-3333) and a message (sms). Figure 4 shows the Smali opcode for the malware activity with the population elements highlighted (*e.g.*, *const-string*, *invoke-virtual*).

Based on Figure 2, we develop the occurrence probability of the population element to build P set as follows in Table 2. We also show the smoothed probability values due to missing elements (f_2, f_4). Here, *e* is a very small number having the value of 0.00001. Similarly Table 3 computes *Q* set based on the repackaged opcode with necessary smoothing.

Table 3. Occurrence of Population Element from Repackaged Malware Application (*Q*)

Opcode	Occurrence (q_i)	Smoothed (q_i')
invoke-super (f_1)	1/8	1/8 - e/8
invoke-static (f_2)	1/8	1/8 – e/8
invoke-direct (f_3)	1/8	1/8 - e/8
invoke-virtual (f_4)	2/8	2/8 - 2e/8
const-string (f_5)	3/8	3/8 - 3e/8
new-instance (f_6)	0/8	2e/8
move-result (f_7)	0/8	2e/8
return-object (f_8)	0/8	2e/8
throw (f_9)	0/8	2e/8

Now, Table 4 shows the detailed steps of computing KLD (*P'*, *Q'*). The last row shows the value of KLD as 0.85367, which we can consider high.

Table 4. Computation of KLD (*P'*, *Q'*)

Element (i)	p'_i	q'_i	$log_2(p'_i)$	$log_2(q'_i)$	$p'_i * log_2(p'_i/q'_i)$
invoke-super (f_1)	0.25000	0.12500	-2.00001	-3.00001	0.25000
invoke-static (f_2)	0.00001	0.12500	-17.60964	-3.00001	-0.00007
invoke-direct (f_3)	0.25000	0.12500	-2.00001	-3.00001	0.25000
invoke-virtual (f_4)	0.50000	0.25000	-1.00001	-2.00001	0.50000
const-string (f_5)	0.25000	0.37500	-2.00001	-1.41505	-0.14624
new-instance (f_6)	0.000002	0.000003	-18.93157	-18.60964	-0.000001
move-result (f_7)	0.000002	0.000003	-18.93157	-18.60964	-0.000001
return-object (f_8)	0.000002	0.000003	-18.93157	-18.60964	-0.000001
throw (f_9)	0.000002	0.000003	-18.93157	-18.60964	-0.000001
Sum ($p'_i * log_2(p'_i/q'_i)$)					0.853722

4. Evaluation

We evaluated our approach by using a set of malware samples obtained from the authors of "Dissecting android malware: Characterization and evolution" (Malgenome project dataset) [28]. The same benchmark has been widely used for related research work as well. From the benchmark, we randomly choose samples from seven malware families: Asroot, CruseWin, *DroidDream*, *Gone60*, Kmin, Nickyspy, and *Plankton*. Table 5 shows brief characteristics of the malware family along with number of samples we evaluate from each of the families. For example, *DroidDram* malware attempts to obtain product ID, device type, language, country, and send them to a remote server via text message. The Asroot steals personal information such as IMEI, phone number and send them to a command and control server via SMS messaging service. The Nickyspy malware applications steals phones state such as call logs besides passing GPS, country code, account information to a remote command and control server.

Table 5. Occurrence of Population Element from Repackaged Malware Application (*Q*)

Malware family	Characteristics	# of samples
Asroot	Steal personal information: IMEI, IMSI and phone number Send information to A C&C server (SMS messenger)	**8**
CruseWin	Send information to A C&C server (SMS messenger) Send to premium-rate SMS messages	**2**
DroidDream	• Steal IMEI, IMSI, and phone number • Send information to remote server via SMS messages	16

Gone60	• Steal IMEI, IMSI and phone number • Steal phone's state: calls log, SMS, contacts, account • Send information to remote server via SMS messages	9
Kmin	• Steal personal information: IMEI, IMSI and phone number • Steal Net information: history and bookmarks, APN, IP, Mac • Send information to A C&C server (SMS messenger)	52
Nickyspy	• Steal phone's state: calls log, SMS, contacts, account • Stolen location information: GPS, Google, Country code • Send information to A C&C server (SMS messenger)	2
Plankton	• Change or copy file in external storage • Download and install apps • Stolen location information: GPS, Google, Country code • Send information to remote server via SMS messages	11

Figure 5 shows an example snapshot of deassembling using the apktool. We then search if original application for a given malware family is available in the Google market place or any sources or not. Since all the legitimate applications infected with the three malware malware have been removed already from market place, we build a small benign Android application (tip calculator that computes tip amount based on user supplied inputs). We dissemble it and compute the *P* set. We then use the same apktool and inject the malware classes and add needed opcode to trigger the classes, and then compute *Q* set from the new application package. We implement a Java class to automate the computing of population set occurrence probability along with KLD computation.

```
PS C:\Users\Administrator\Desktop\Android Malware-Repackaging\apktool-install-windows-r05-ib

I: Using Apktool 2.0.0-RC2 on dc.apk
I: Loading resource table...
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: C:\Users\Administrator\apktool\framework\1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
testCleaning up unclosed ZipFile for archive C:\Users\Administrator\apktool\framework\1.apk
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
PS C:\Users\Administrator\Desktop\Android Malware-Repackaging\apktool-install-windows-r05-ib
```

Figure 5. Example Run of Apktool for Decompling Malware

Figure 6 shows an example of Smali code from a sample of Plankton malware application. Here, the opcode **invoke-virtual** (highlighted) are used to read information from an input stream and send it to an output file stream, which was part of downloading application from network stream to a local storage of the Android device.

```
.line 163
 .local v0, "b":[B
 invoke-virtual {v3, v0}, Ljava/io/InputStream;->read([B)I
 .line 164
 invoke-virtual {v2, v0}, Ljava/io/FileOutputStream;-
>write([B)V
```

Figure 6. An Example of Smali Code from a Plankton Sample

We discuss the obtained results now for selected malware families. Figure 7 shows the population element distribution (f_1-f_9) for all 16 samples of the DreamDroid. Figure 8 shows the population element distribution (f_1-f_9) for all 16 samples of the DreamDroid family after they are injected into *P*. For each of the repackaged applications, the occurrence of population elements (f_1-f_9) increases significantly compared to the legitimate application.

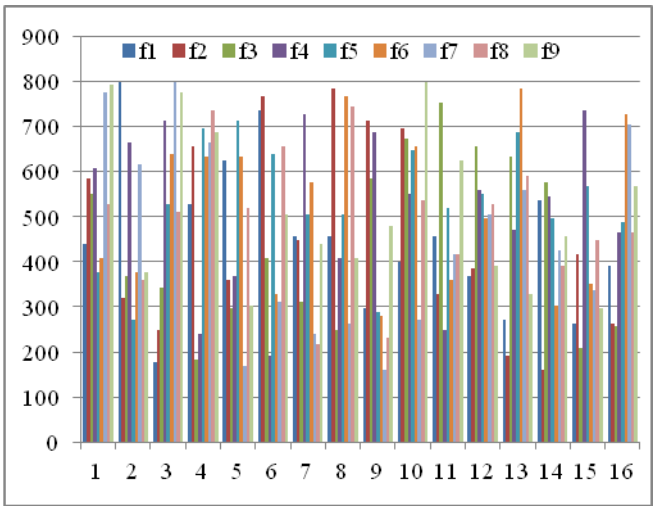

Figure 7. Population Distribution of DreamDroid Samples (P)

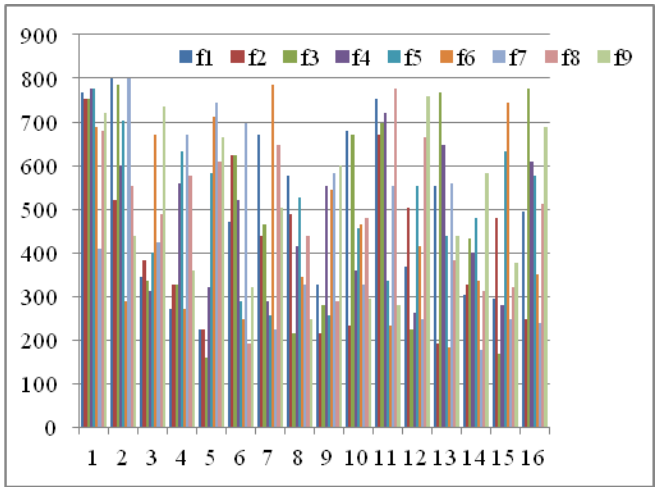

Figure 8. Population Distribution of DreamDroid Samples (Q)

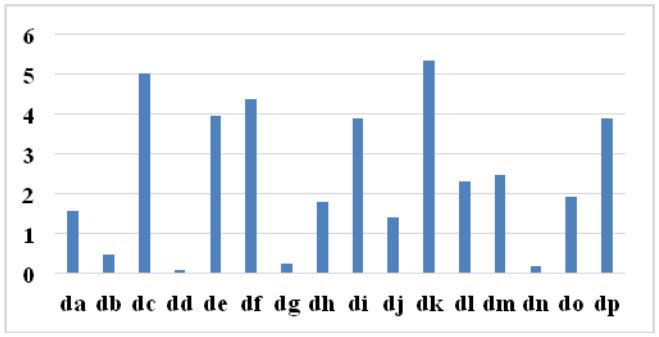

Figure 9. KLD Values for Various DreamDroid Samples

Figure 9 shows KLD (*P*, *Q*) values as part of detecting repackaged applications having *DreamDroid* malware. We rename 16 different malware samples from *da-dp* for reader's convenience. Note that the highest KLD we observed was for the dc sample (5.3382) and the lowest value was for the dd sample (0.1003).

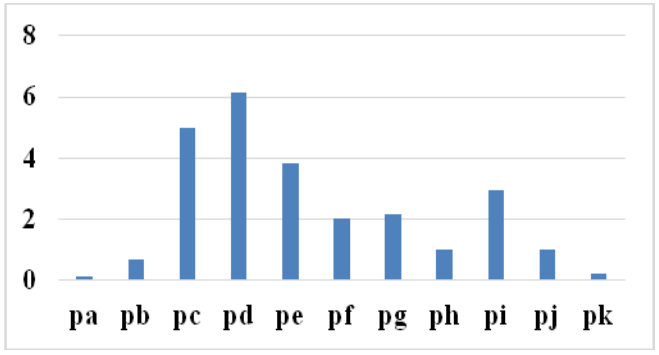

Figure 10. KLD Values for Plankton Family Samples

Similarly, Figures 10 and 10 display the obtained KLD values as we detect the repackaged malware from *Plankton* and *Gone60* family. The highest and lowest KLD values for Plankton family were 6.1368 0.01968 and, respectively. The highest and lowest KLD values for *Gone60* family were 4.1396 and 0.01484, respectively.

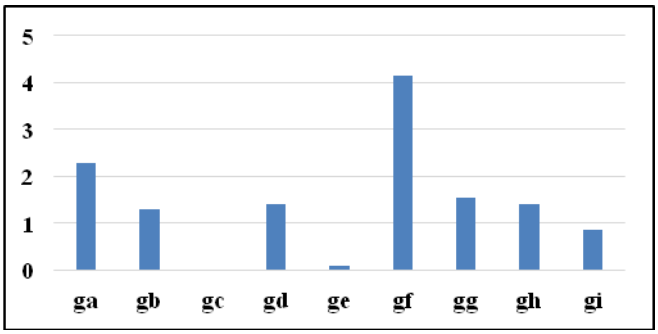

Figure 11. KLD Values for Malware Samples of Gone60

Figure 11 shows the KLD values for *Asroot* family malware. The highest and lowest KLD values for *Asroot* family were 5.67 and 1.42, respectively.

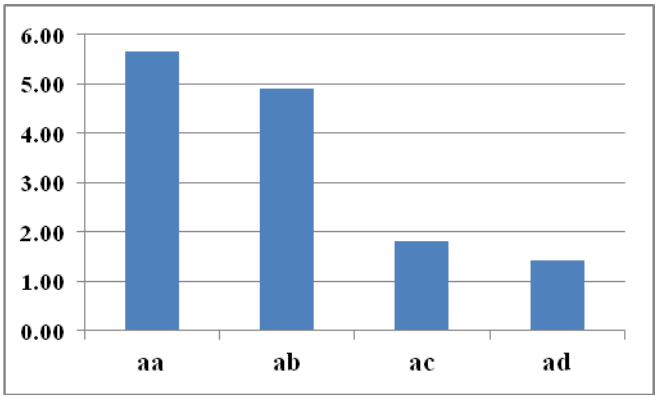

Figure 12. KLD Values for Malware Samples of Asroot

From the obtained data, a KDL threshold value can be chosen to detect new repackaged application. We suggest choosing KLD values above zero.

5. Conclusions and Future Work

This paper proposes a metric-based approach using Kullback-Leibler Distance (KLD) to identify repackaged Android applications. We propose a set of population element features to compute the occurrence probability of specific Smali opcode that may indicate likely malicious activities such as method call invocation performing a message sending operation. The approach has been evaluated with a set of sample malware applications from a real-world benchmark suite and the initial obtained results look promising. For all the repackaging cases, we find that

the KLD value exceeds zero and a higher number frequently appears. Our future works remains to include more elements for population building set as well as validating with more sample malwares. We plan to develop a network-based online detection approach to validate whether an application is repackaged or not for a given legitimate application based on other suitable metrics.

6. References

[1] N. Lomas, Android Still Growing Market Share By Winning First Time Smartphone Users, May 2014, Accessed from from http://techcrunch.com/2014/05/06/android-still-growing-market-share-by-winning-first-time-smartphone-users/

[2] Y. Zhou and X. Jiang, "Dissecting Android Malware: Characterization and Evolution," Proc. of IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 2012, pp. 95-109.

[3] Smali opcode, Acessed from http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html

[4] Repackaged Applications, Blog report from University of Lousiana at Lafayette, Accessed from http://ulsrl.org/repackaged-applications/

[5] Dex2jar, Accessed from https://code.google.com/p/dex2jar/

[6] A. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A Survey of Mobile Malware in the Wild," *Proc. of the ACM Workshop Security and Privacy in Mobile Devices (SPMD)*, 2011, pp. 3-14.

[7] A. Amamra, C. Talhi, and J. Robert, "Smartphone Malware Detection: From a Survey Towards Taxonomy," *Proc. of 7th International Conference on Malicious and Unwanted Software (MALWARE)*, October 2012, Puerto Rico, USA, pp. 79-86.

[8] V. Cooper, H. Shahriar, and H. Haddad, "A Survey of Android Malware Characteristics and Mitigation Techniques," *Proc. of the 11th International Conference on Information Technology: New Generations*, IEEE CPS, Las Vegas, USA, April 2014, pp. 327-332.

[9] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A Study of Android Application Security," *Proc. of USENIX Security Symposium*, August 2011.

[10] W. Enck, M. Ongtang, and P. McDaniel, "On Lightweight Mobile Phone Application Certification," *Proc. 16th ACM Conf. Computer and Communications Security (CCS 09)*, ACM, 2009, pp. 235-245.

[11] L. Batyuk, M. Herpich, S. Camtepe, K. Raddatz, A. Schmidt, and S. Albayrak, "Using Static Analysis for Automatic Assessment and Mitigation of Unwanted and Malicious Activities within Android Applications," *Proc. of 6th International Conference on Malicious and Unwanted Software (MALWARE)*, October 20011, pp. 66-72.

[12] D. Barrera, H. Kayacik, P. Oorchot, and A. Somayaji, "A Methodology for Empirical Analysis of Permission-Based Security Models and Its Application to Android," *Proc. 17th ACM Conf. Computer and Communications Security (CCS)*, 2010, pp. 73-84.

[13] A. Felt, K. Greenwood, and D. Wagner, "The Effectiveness of Application Permissions," *Proc. of the 2nd USENIX Conference on Web Application Development (WebApps)*, 2011.

[14] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth, "TaintDroid: An Information-Flow Tracking System for Real-Time Privacy Monitoring on Smartphones," *Proc. 9th USENIX Symposium Operating Systems Design and Implementation*, 2010.

[15] T. Blasing, L. Batyuk, A. Schmidt, S. Camtepe, & S. Albayrak, "An Android Application Sandbox System for Suspicious Software Detection," *Proc. of 5th IEEE Malicious and Unwanted Software*, 2010, pp. 55-62.

[16] C. Yang, V. Yegneswaran, P. Porras, and G. Gu, "Detecting Money-Stealing Apps in Alternative Android Markets," *Proc. of the 2012 ACM Conference on Computer and Communications Security (CCS)*, October 2012, Raleigh, North Carolina, USA, pp. 1034-1036.

[17] Y. Zhou and X. Jian, "Detecting Passive Content Leaks and Pollution in Android Applications," Proc. of 20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013.

[18] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic Detection of Capability Leaks in Stock Android Smartphones," In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, NDSS, 2012.

[19] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry, "Towards Taming Privilege-Escalation Attacks on Android," *Proc. of the 19th Annual Symposium on Network and Distributed System Security*, NDSS, 2012.

[20] Java decompiler, Accessed from http://jd.benow.ca/

[21] Android-apktool, https://code.google.com/p/android-apktool/

[22] J. Crussell, C. Gibler, and H. Chen. Attack of the clones: Detecting cloned applications on android markets. *Proc. of ESORICS*, pages 37–54, 2012.

[23] S. Li. Juxtapp: A scalable system for detecting code reuse among android applications. Master's thesis, EECS Department, University of California, Berkeley, May 2012. http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-111.html

[24] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," *Proc. of the 2nd ACM conference on Data and Application Security and Privacy (CODASPY)*, pp. 317–326.

[25] T. Cover and J. Thomas, *Elements of Information Theory*, John Wiley and Sons, 2006.

[26] Brigitte Bigi, "Using Kullback-Leibler Distance for Text Categorization," *Lecture Notes in Computer Science (LNCS)*, Volume 2633, 2003, pp. 305-319.

[27] H. Huang, S. Zhu, P. Liu, and D. Wu, "A Framework for Evaluating Mobile App Repackaging Detection Algorithms," Trust and Trustworthy Computing, *Lecture Notes in Computer Science*, Volume 7904, 2013, pp 169-186.

[28] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," Proc. of IEEE Symposium on Security and Privacy (SP), Oakland, CA, USA, May 2012, pp. 95–109.

[29] H. Thanh, "Analysis of Malware Families on Android Mobiles: Detection Characteristics Recognizable by Ordinary Phone Users and How to Fix It," *Journal of Information Security*, 2013, 4, pp. 213-224.

